IN THE SPECIFICATION:

At page 1, before line 4, please insert the following:

— This application is a continuation of U.S. patent application Serial No. 09/483,123 filed January 14, 2000, entitled A PROGRAM-DEVELOPMENT ENVIRONMENT FOR USE IN GENERATING APPLICATION PROGRAMS, now U.S. Patent No. __________. The entirety of said U.S. Patent No. _________ is hereby incorporated by reference. —

Please amend the four full paragraphs starting at page 1, line 5 as shown below:

U.S. Patent Application Serial No. 09/483,760 [(130017-0009)] entitled, METHOD AND APPARATUS FOR RESOLVING DIVERGENT PATHS IN GRAPHICAL PROGRAMMING ENVIRONMENTS, filed January 14, 2000, now Pat. No. 6,425,121;

U.S. Patent Application Serial No. 09/483,759 [(130017-0010)] entitled METHOD AND APPARATUS FOR DETECTING AND RESOLVING CIRCULAR FLOW PATHS IN GRAPHICAL PROGRAMMING SYSTEMS, filed January 14, 2000;

U.S. Patent Application Serial No. 09/483,122 [(130017-0011)] entitled, REPEATING PROGRAM OBJECT FOR USE WITH A GRAPHICAL PROGRAM-DEVELOPMENT SYSTEM, filed January 14, 2000, now U.S. Pat. No. 6,425,120; and

U.S. Patent Application Serial No. 09/483,124 [(130017-0012)] entitled, PROGRAM OBJECT FOR USE IN GENERATING APPLICATION PROGRAMS, filed January 14, 2000.

Please amend the carry over paragraph between pages 3-4 as shown below:

Fig. 1 illustrates a conventional Visual Basic work space 100 that may be displayed on a computer screen. The work space 100 includes a Form window 102 and a tool palette 104. The tool palette 104 contains a plurality of icons, which represent individual controls, including a vertical scroll control 106 and a text label control 108, among others. A developer may select any of the controls contained on palette 104 to cause the selected control to appear on the Form window 102. By selecting the vertical scroll icon 106, for example, a corresponding vertical scroll image 110 is displayed on the Form window 102. A text label image 112 may be placed on the Form window 102 in a similar manner. At this point, however, there is no inter-relationship between the objects corresponding to vertical scroll image 110 and text label image 112. In order to establish some such relationship (e.g., causing the text label to display the current position of the vertical scroll), the developer must write a subroutine (e.g., an event handler). Each line or statement of the subroutine, moreover, must conform to the syntax and keyword commands of the underlying programming language (e.g., Visual Basic). Specifically, the developer selects the vertical scroll 110, thereby causing a code window 114 to be displayed on screen 100. Inside the code window 114 [144], the developer writes a text-based subroutine 116 that causes the output of the vertical scroll 110 to be displayed in the text label 112.

Please amend the first full paragraph on page 4 as shown below.

When this program is subsequently run, images for the vertical scroll bar 110 and the text label 112 will appear on the screen of the user as part of a user interface. The text label 112 [110], moreover, will display the position of the vertical scroll bar 110 (e.g., "2256"). If the user moves the slider bar of the vertical scroll, the contents of text label change to display the scroll bar's new position (e.g., "3891"). As shown, with Visual Basic, the developer need not "write" any code to cause the vertical scroll bar image 110 or the text label image 112 to be displayed on the computer screen during run time. In addition, during the programming phase, the developer may move and re-size these user interface elements simply by manipulating their appearance on the Form window 102 (e.g., with a mouse) in a conventional manner. Due to the relative ease with which application programs having user interface elements can be created, Visual Basic has become a highly popular programming tool. However, in order to develop a meaningful application program (i.e., one in which there is some inter-relationship between the user interface elements), the developer must write, in a text-based format, one or more sub-routines. Thus, the developer must learn and is limited by the syntax and keyword structures of Visual Basic.

Please amend the second full paragraph on page 10 as shown below:

In accordance with the present invention, a program-development environment 310 is also executing on the computer system 200. The program-development environment 310 includes an extensible visual programming system 312 and a graphical designer system 314. The visual programming system 312, in turn, may include an extensibility object 316, which provides an interface for communication between the programming system 312 and the graphical designer system 314 as indicated by arrows 318 and 320. Arrow 320 represents calls from the designer system 314 to the programming system 312, while arrow 318 represents calls from the programming system 312 to the designer system 314. Additionally, both the graphical designer system 314 and the visual programming system 312 may communicate directly with the operating system 302, e.g., exchange task commands and windows events, via API layer <u>304</u> [308], as indicated by arrows 322-328.

Please amend the second full paragraph on page 18 as shown below:

The developer then selects the next program object or control for use in the application program being created. Suppose that the developer selects the label icon 432 (Fig. 4B) from the User Interface sub-toolbar 414c. As shown in Fig. 4C, the program-development environment 310, in response, causes a symbolic representation 434 of a label program object to appear in designer window 406. Symbolic representation 434 also includes a plurality of terminals 436a-c, and may further include a name 434a. The program-development environment 310 additionally directs the visual programming system 304 to add a label program object to form window 404. Since the label program object is also a user interface element, like the vertical scroll bar, the visual programming system 304 additionally causes a label image 438 to be drawn on the user form object 408.

Please amend the first full paragraph on page 21 as shown below:

Specifically, the graphical designer system 314 directs the visual programming system 312 through calls to its extensibility object 316, as arrow 320 indicates, to instantiate a wire component control or program object from the wire object class and to add this object to the form window 404. That is, form window 404 adds a pointer to the wire program object to its linked list of controls. It should be understood that the wire construct 440 [400] appearing in the designer window 406 is preferably just a symbolic representation of the wire program object added to the form window 404. The visual programming system 312 also assigns a name to this program object, e.g., Wire2, which it returns to the designer system 314. As described below, as part of its initialization procedure, designer system 314 preferably directs the visual programming system 312 to instantiate and add a wire program object, which may be named Wire1, to the form window 404. Thus, the "first" wire that is drawn on the designer window 406 by the developer actually corresponds to the second wire program object to be instantiated and added to the form window 404. Therefore, this wire program object is typically assigned the name Wire2.

Please amend the carry over paragraph between pages 24-25 as shown below:

After identifying the source and sink control objects, the designer system 314 is ready to set the Sink, Source and Trigger properties 442h, 442i and 442m of Wire2. The wire program object's Source property is preferably a concatenation of the following information: the name of the form window 404 on which the source program object resides, e.g., Form1, the name of the source program object, e.g., VScrollBar1, and the property associated with the linked terminal at the source program object, e.g., Value. The Source property may further be concatenated with the event associated with the linked terminal at the source program object, e.g., DataReady. The designer system 314 preferably obtains the source event and property parameters for use in setting the wire's Source property from the event field 502 and property field 504 from the terminal data structure 500 associated with linked terminal at the source program object, i.e., terminal 430c. For data output type terminals, such as terminal 430c, system 314 similarly obtains the Source-Group property parameter 442j from the group identifier field 508 [805] from the corresponding terminal data structure 500.

Please amend the first full paragraph on page 38 as shown below:

Figs. 10A-10B illustrate a flow chart of the steps corresponding to the running of an application program whose flow diagram includes one or more forks, such as diagram 900 (Fig. 9). In response to the occurrence of its corresponding triggering event, a wire object that is part of any flow diagram, first determines whether it is at a fork as indicated at step 1002. As described above, by virtue of the initialization process, all wire objects know whether or not they are at a fork of the flow diagram. If it is at a fork, the triggered wire object invalidates its sink property and directs the other wire objects at the fork to invalidate their sink properties as indicated at block 1004. In response to having a control or data input property invalidated, a program object preferably issues an InvalidateGroup event as indicated at block 1006. Those wire objects connected to the data output or control output terminals of such a program object are configured to respond to the issuance of the InvalidateGroup event. In other words, these wire objects register with their source objects, preferably using the Event_Advise_Notification method, so as to learn of any InvalidateGroup events. These wire objects then respond by invalidating their own respective sink properties as indicated at block 1008. As indicated at block 1009, the steps of blocks 1004-1008 are preferably repeated by the remaining wire and other program objects whose symbolic representations are located downstream of the fork relative to the root, thereby invalidating the respective data and control input terminals.

Please amend the first two full paragraphs on page 43 as shown below:

When the program object is triggered, it first increments the busy indicator (e.g., the counter), preferably by "1", as indicated by block 1208. The object then determines whether the value of its busy counter exceeds some predetermined threshold as indicated by decision block 1210. In the preferred embodiment, the threshold is set to "1". If the busy counter does not exceed the threshold, the object then executes its respective function as indicated by No arrow 1212 leading to block 1214, which preferably corresponds to steps 704-718 of Fig. 7 described above. That is, the object places its output data on its data output terminal and issues its DataReady and its ControlOut events. As described above, other objects, such as wire program objects, may respond to the DataReady and ControlOut events. As indicated at decision block 1216 and No arrow 1218 which loops back to block 1216, the object waits until such "observers" have returned from the DataReady and ~~Control out~~ ControlOut events.

Specifically, upon issuing its DataReady event, program control ~~(e.g.,~~ e.g., processing by the CPU 210 (Fig. 2), shifts from executing the steps of the program object to executing the steps corresponding to the event handler procedure that is triggered by the object's DataReady event. When this event handler procedure is finished, program control then returns to the object so that its execution may be completed. So that program control may be returned to the appropriate location within the steps corresponding to the program object, a pointer to the location is typically pushed onto a stack within memory 214 (Fig. 2). When the event handler procedure or code has finished executing, this pointer is popped off of the stack and processing resumes at the appropriate location. As described above, execution of the event handler procedure or code may result in the issuance of one or more events (e.g., Action, Done, etc.) and it may be interrupted so that program instructions or code triggered by these events may be executed.

Please amend the first three paragraphs on page 46 as shown below:

Do

{~~[~~statements~~]~~}

Loop Until/While *condition*.

where "{~~[~~statements~~]~~}" are the particular code statements that are to be repeated and *condition* refers to the condition that stops the loop. A Loop Until *condition* checks the *condition* after running through the loop and repeats only if the *condition* is FALSE. A Loop While *condition* also checks the *condition* after running through the loop but repeats only if the *condition* is TRUE. Do Loops may also be configured to check the condition before entering the loop.

Please amend the two paragraphs starting at line 14 of page 46 as shown below:

For *counter* = *start* To *end* {~~[~~Step *increment*~~]~~}

{~~[~~statements~~]~~}

Next {~~[~~*counter*~~]~~}

Please amend the first full paragraph on page 47 as shown below:

Fig. 13 is a preferred representation of a GUI 1300 generated by the program-development environment 310 (Fig. 3) on computer screen 235 similar to GUI 400 (Fig. 4A) described above. The Core sub-toolbar 414b of the designer window 406 includes a plurality of icons that correspond to program objects for use in performing loop-type functions, among others, within the application program being developed. In particular, sub-toolbar <u>414b</u> [414a] includes icons for a For Loop, a Do Loop, a Timer, and a Break control, among others. Each of these icons, in a similar manner as described above, corresponds to an object class from which one or more program objects or controls may be instantiated. As described herein, these loop controls can generate multiple outputs from a single input.